Selected Topics on Java Programming

Unit Testing with Junit 4. Java DataBase Connectivity (JDBC). Generics. Java 8 Stream API

Where to Find The Code and Materials?

https://github.com/iproduct/course-java-web-development

Бази от данни

Съдържание

1. Бази от данни (БД). Видове БД.
2. Системи за управление на бази от данни (СУБД)
3. Релационен и обектен модели
4. Релации и релационни схеми
5. Базови и производни релации. Домейни. Ограничения
6. Ключове.
7. Връзки между таблици и кардиналност.
8. Операции над релационни бази от данни
9. Нормализация на бази от данни
10. Транзакции и конкурентност
11. Новости в JDBC™ 4.1 (Java 7): try-with-resources и RowSets

Бази от данни (БД)

- Дефиниция (Wikipedia):

База данни (БД, още база от данни) представлява колекция от логически свързани данни в конкретна предметна област, които са структурирани по определен начин. В първоначалния смисъл на понятието, използван в компютърната индустрия, базата от данни се състои от записи, подредени систематично, така че компютърна програма да може да извлича информация по зададени критерии.

Видове БД – според структурата

- Йерархични бази от данни
  - директорийна структура, файлова система
  - IBM IMS – 1968 г.
- Мрежови модел на БД - Чарлс Бейчман
  - позволява представяне на връзки 1:N между различните нива на йерархията
  - CODASYL IDMS – 1971 г.
- Релационни БД – Едгар Код – 1970 г.
- Обектно-ориентирани бази от данни

Видове БД – според предназначението

- Оперативни БД
- Аналитични БД
- Data warehouse
- БД за крайни потребители
- Външни БД
- Хипермедийни БД
- Навигационни БД
- Документно-ориентирани БД
- БД работещи в реално време

Системи за управление на бази от данни (СУБД)

- **Def:** съвкупност от компютърни програми, използвани за изграждане, поддръжка и използване на бази от данни
- **Примери:** MySQL, PostgreSQL, DB2, Microsoft SQL Server, Access, Oracle, Paradox, dBase, FoxPro, Cliper, Sybase, Informix
- **СУБД** създават, обработват и поддържат определени структури от данни. Най-популярен е релационният модел, при който данните се организират в таблици, между които се осъществяват връзки (т.н. релации). Таблиците се състоят от именувани редове и колони. Редовете се наричат записи, а колоните - полета.

Основни компоненти на СУБД

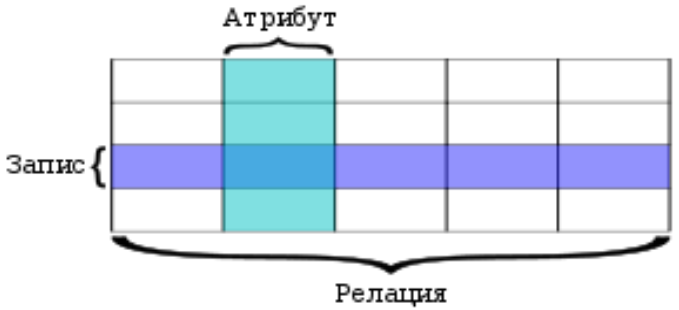
- Релационни СУБД - RDBMS
  - Интерфейсни драйвери
  - SQL engine
  - Transaction engine
  - Relational engine
  - Storage engine
- Обектно-ориентирани БД – ODBMS
  - Езикови драйвери – C++, Java, .Net, Ruby
  - OO език за заявки – JPAQL, LINQ, ...
  - Transaction & Storage engines

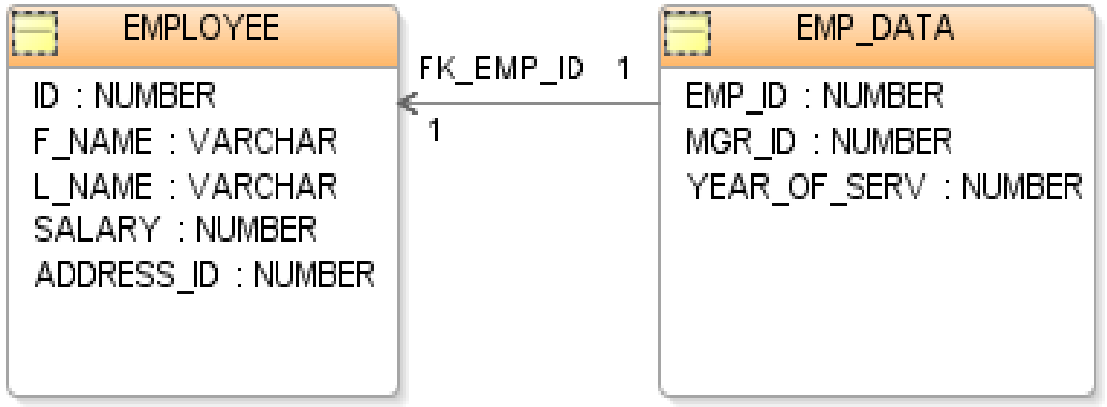
Релационна база от данни

- **Def:** Релационна база данни е тип база данни, която съхранява множество данни във вид на релации, съставени от записи и атрибути (полета) и възприемани от потребителите като таблици.
- Релационните бази данни понастоящем преобладават при избора на модел за съхранение на финансови, производствени, лични и други видове данни.
- Терминът „релационна база данни“ за първи път е предложен през 1970 година от Едгар Код, учен в IBM.

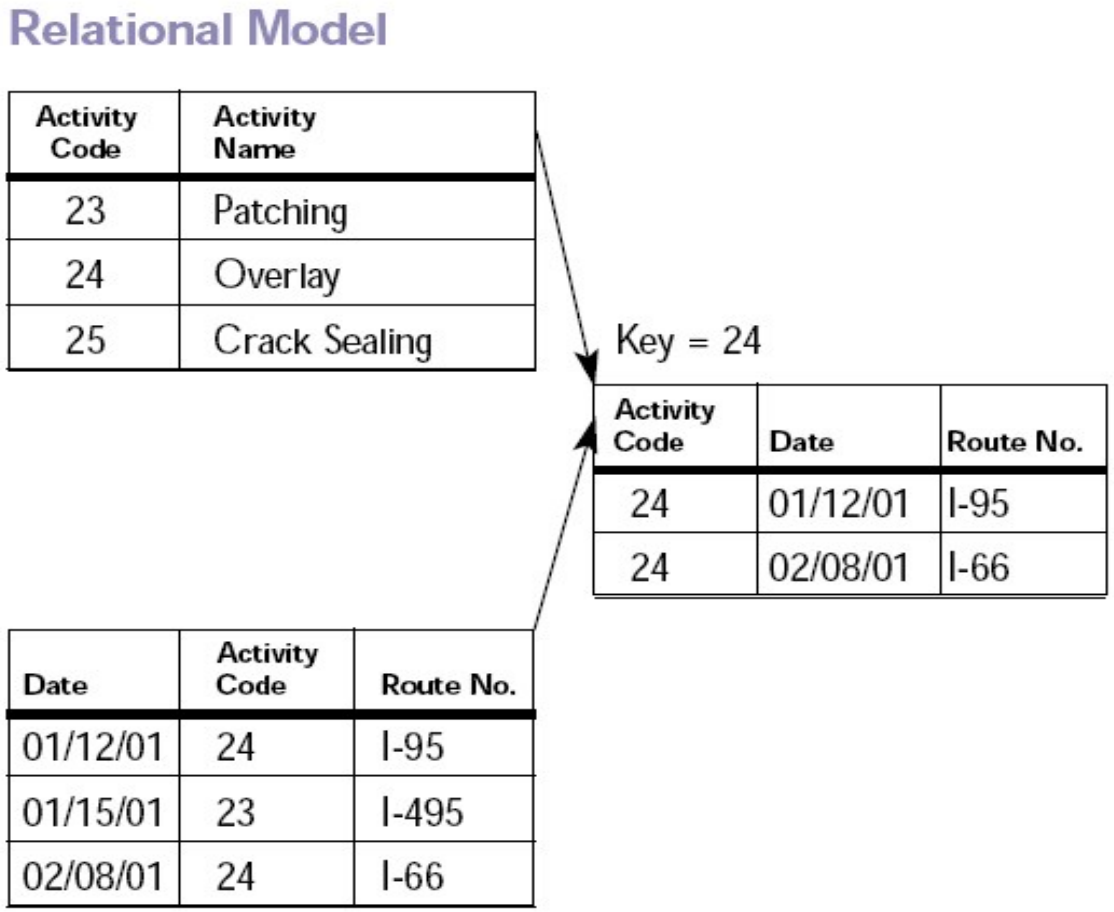
Релационен модел

- релация, релационна схема (relation) ↔ таблица (table),
- запис, кортеж (tuple) ↔ ред (row)
- атрибут, поле (attribute) ↔ стълб, колона (column)

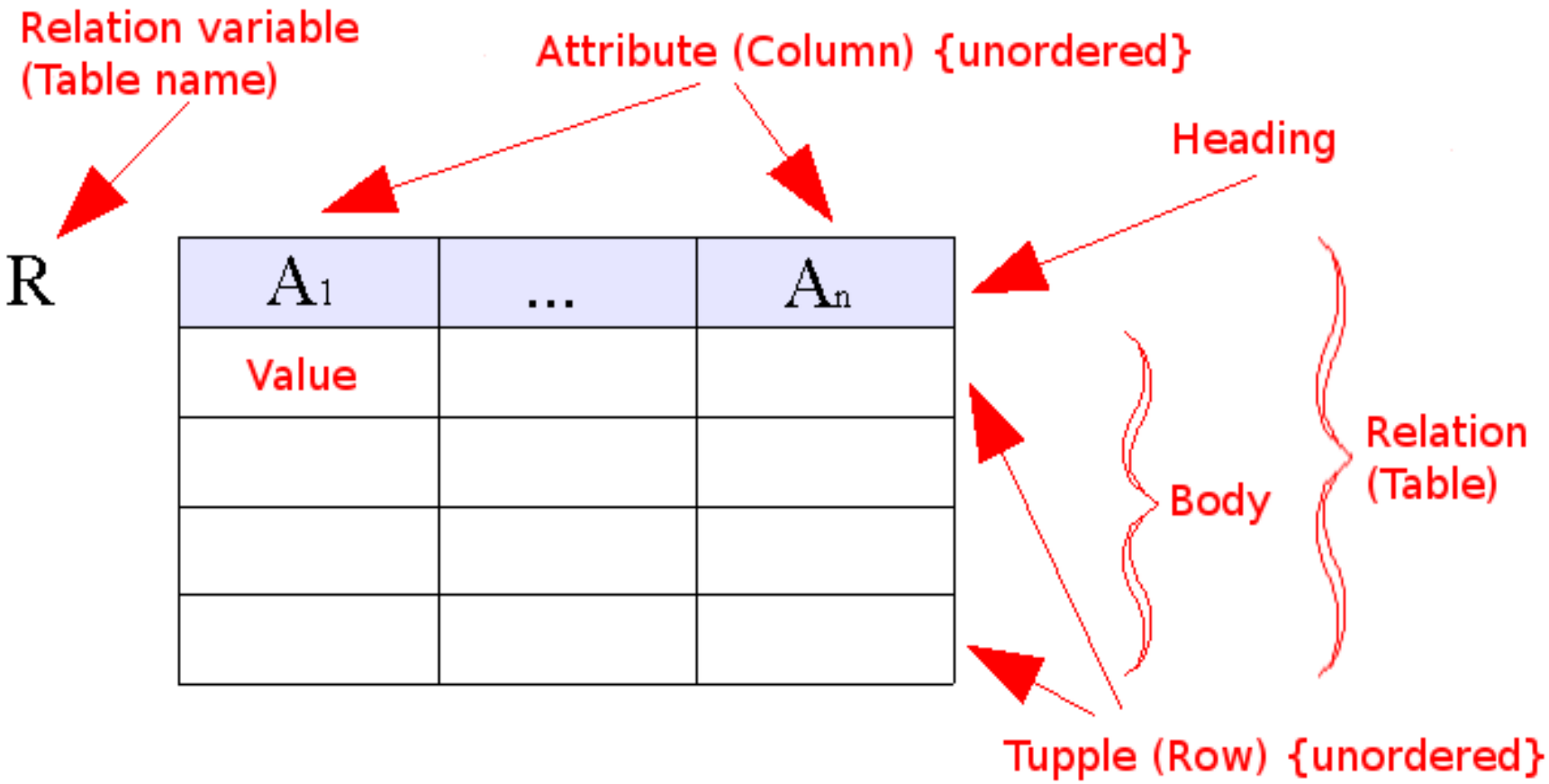




Релационен модел



Релационен модел



Релации и релационни схеми

- **Def:** Релацията (relation) се дефинира като множество от записи, които имат едни и същи атрибути. Записът обикновено представя обект и информация за обекта, който обичайно е физически обект или понятие. Релацията обикновено се оформя като таблица, организирана по редове и колони. Всички данни, които се съдържат в даден атрибут, принадлежат на едно и също множество от допустими стойности, наречено домейн, и съблюдават едни и същи ограничения.
- Заглавието (heading) на таблицата се нарича релационна схема, а множеството от всички релационни схеми в БД – схема на БД (database shema)

Базови и производни релации. Домейни. Ограничения

- **Def:** . Релациите, които съхраняват данните, се наричат **базови релации (base relations)** или таблици (**tables**). Други релации обаче не съхраняват данни, а се изчисляват чрез прилагането на операции над други релации. Наричат се **производни релации (отношения),** а в приложенията за бази данни се наричат заявка (**query**) и изглед (**view**).

- **Def:** **Домейн** в базите данни означава множеството от допустимите стойности на даден атрибут на релация, т.е. представлява известно ограничение върху стойностите на атрибута.

- **Def:** **Ограниченията (constraints)** позволяват в още по-голяма степен да се специфицират стойностите, които атрибутите от даден домейн могат да приемат – например от 1 до 10.

Ключове

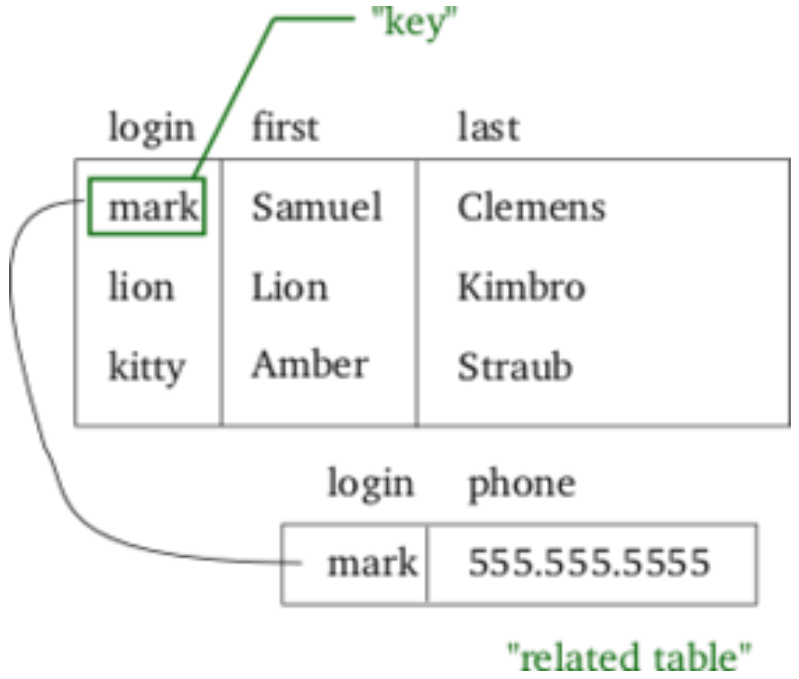
- **Ключ (key)** се наричат един или повече атрибута, такива че:

  1) релацията няма две различни записа с едни и същи стойности за тези атрибути

  2)няма строго подмножество на тези атрибути с горното свойство

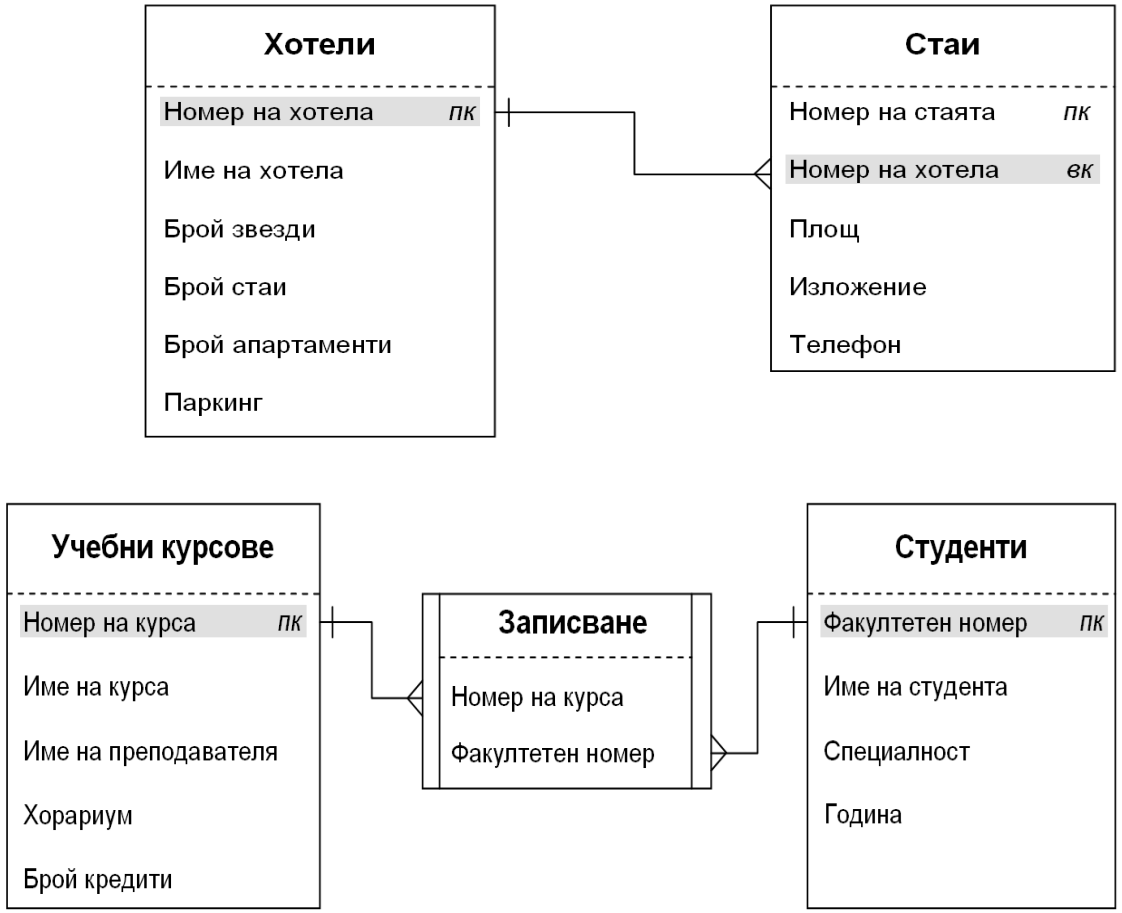
- **Първичен ключ (primary key)** е атрибут (по-рядко група атрибути), който служи да идентифицира по уникален начин всеки запис (екземпляр) на релацията

- **Външният ключ (foreign key)** е необходим, когато налице е отношение между две таблици (релации).



Връзки между таблици и кардиналност

- **Отношение (relationship**) се нарича зависимост, съществуваща между две таблици, когато записи от първата таблица могат да се свържат по някакъв начин със записи от втората таблица.
- Кардиналност:
  - едно към едно (1:1),
  - едно към много (1:N),
  - много към много (M:N).



Операции над релационни бази от данни

- **Оператор обединение (union)** комбинира записи от две релации и премахва от резултата всички евентуални повтарящи се записи. Релационният оператор обединение е еквивалентен на SQL-оператора UNION.
- **Оператор сечение (intersection)** извежда множеството от записи, които са общи за двете релации. Сечението в SQL е реализирано чрез оператора INTERSECT.
- **Оператор разлика (difference)** се прилага над две релации и в резултат връща множеството от записите от първата релация, които не съществуват във втората релация. В SQL разликата е имплементирана посредством оператора EXCEPT или MINUS.

Операции над релационни бази от данни

- **Оператор декартово произведение** или само произведение (Cartesian product, cross join, cross product) на две релации представлява съединение (join), при което всеки запис от първата релация се конкатенира с всеки запис от втората релация. В SQL операторът е реализиран под името CROSS JOIN.
- **Операцията селекция (selection, restriction)** връща само онези записи от дадена релация, които отговарят на избрани критерии, т.е. подмножество в термините на теорията на множествата. Еквивалентът на селекцията в SQL е заявка SELECT с клауза WHERE.

Операции над релационни бази от данни

- **Операцията проекция (projection)** е по своя смисъл селекция, при която повтарящите се записи се отстраняват от резултата. В SQL е реализирана с клаузата GROUP BY или чрез ключовата дума DISTINCT, внедрена в някои диалекти на SQL.

- **Операцията съединение (join)**, дефинирана за релационни бази данни, често се нарича и естествено съединение (natural join). При този вид съединение две релации са свързани посредством общите им атрибути. В SQL тази операция е реализирана приблизително чрез оператора за съединение INNER JOIN. Други видове съединение са лявото и дясното външни съединения, внедрени в SQL като LEFT JOIN и RIGHT JOIN, съответно.

Операции над релационни бази от данни

- **Операцията деление (division)** е малко по-сложна операция, при която записи от една релация в ролята на делител се използват, за да се раздели втора релация в ролята на делимо. По смисъла си, тази операция е обратна на операцията (декартово) произведение.

Нормализация на бази от данни

- **Нормализацията**, т.е. привеждането в нормална форма включва набор от практики по отстраняването на повторения сред данните, което от една страна води до икономия на памет и повишено бързодействие, а от друга страна предпазва от аномалии при манипулирането с данните (вмъкване, актуализиране и изтриване) и от загуба на тяхната цялост. В процеса на нормализация се осигурява оптимална структура на базата от данни, основаваща се на взаимозависимостта между данните. Структурата на таблиците се трансформира, с цел да се оптимизират функционалните зависимости на съставните им атрибути.
- **Нормални форми**

Транзакции и конкурентност

- **Транзакция = бизнес събитие**
- **ACID правила:**
  - **Атомарност(Atomicity)**: или се изпълнява цялата транзакция – всички задачи, или не се изпълнява никоя от тях (rolled back).
  - **Съгласуваност (Consistency)**: транзакцията трябва да запазва integrity constraints.
  - **Изолация (Isolation)**: две едновременни транзакции не могат да си взаимодействат.
  - **Постоянство (Durability)**: успешно завършените транзакции не могат да се отменят.

Java DataBase Connectivity (JDBC)

Practical Exercises

Java Database Connectivity (JDBC)

- **Java Database Connectivity** (**JDBC**) е application programming interface (API) на езика Java, който дефинира как клиентите могат да достъпват, извличат и модифицират данни в една релационна база от данни.
- JDBC-to-ODBC bridge позволява връзки към всякакви ODBC-достъпни източници на данни в Java virtual machine (JVM) среда.

Java Database Connectivity (JDBC) – пример (1)

```
Scanner sc = new Scanner(System.in);
Properties props = new Properties();

System.out.println("Enter  username (default root): ");
String user = sc.nextLine().trim();
user = user.length() > 0 ? user : "root";
props.setProperty("user", user);

String password = sc.nextLine().trim();
password = password.length() > 0 ? password : "root";
props.setProperty("password", password);
```

Java Database Connectivity (JDBC) – пример (2)

```
// 1. Load jdbc driver (optional)
try {
        Class.forName("com.mysql.cj.jdbc.Driver");
} catch (ClassNotFoundException e) {
        e.printStackTrace();
        System.exit(0);

}
System.out.println("Driver loaded successfully.");


// 2. Connect to DB
Connection connection =
     DriverManager.getConnection(
       "jdbc:mysql://localhost:3306/employees?useSSL=false", props);

System.out.println("Connected successfully.");
```

Java Database Connectivity (JDBC) – пример (3)

```
// 3. Execute query
PreparedStatement stmt =
    connection.prepareStatement("SELECT * FROM employees JOIN salaries ON
employees.emp_no=salaries.emp_no WHERE salaries.salary > ?");


System.out.println("Enter  minimal salary (default 20000): ");
String salaryStr = sc.nextLine().trim();

double salary = Double.parseDouble(salaryStr);

stmt.setDouble(1, salary);
ResultSet rs = stmt.executeQuery();
```

Java Database Connectivity (JDBC) – пример (4)

```
// 4. Process results
while (rs.next()) {
        System.out.printf("| %-15.15s | %-15.15s | %10.2f |\n",
                rs.getString(2),
                rs.getString("last_name"),
                rs.getDouble("salary")
        );
}

// 5. Close connection and statement
connection.close();
```

Новости в JDBC™ 4.1 (Java 7): try-with-resources

- java.sql.Connection, java.sql.Statement и java.sql.ResultSet имплементират интерфейса **AutoCloseable**:

```
Class.forName("com.mysql.jdbc.Driver");                    //Load MySQL DB driver
try (Connection c = DriverManager.getConnection(dbUrl, user, password);
   Statement s = c.createStatement() ) {
   c.setAutoCommit(false);
   int records = s.executeUpdate("INSERT INTO product "     //Insert new product
            + "VALUES ('CP-00002', 'Lenovo', 790.0, 'br', 'Latop')");
   System.out.println("Successfully inserted "+ records + " records.");
   records = s.executeUpdate("UPDATE product "              //Update product price
            + "SET price=470, description='Classic laptop' "
            + "WHERE code='CP-00001'");
   System.out.println("Successfully updated "+ records + " records.");
   c.commit();                                               //Finish transaction
}
```

Новости в JDBC™ 4.1 (Java 7): RowSets (1)
https://docs.oracle.com/javase/tutorial/jdbc/basics/rowset.html

- **RowSet** – дава възможност да работим с данните от таблиците в базата от данни като с нормални JavaBeans™ компоненти – да достъпваме и променяме стойностите като свойства (properties), да закачаме слушатели на събития свързани с промяна на данните (event listeners), както и да скролираме (scroll) и променяме (update) данните в заредените в RowSet-а редове
- Свързан (connected) RowSet
  - **JdbcRowSet** – обвиващ клас около стандартния JDBC ResultSet
- Несвързан (disconnected) RowSet
  - **CachedRowSet** - кешира данните в паметта, подходящ за изпращане
  - **WebRowSet** - подходящ за изпращане на данните през HTTP (XML)
  - **JoinRowSet** - позволява извършване на JOIN без свързване към БД
  - **FilteredRowSet** - позволява локално филтриране на данните (R/W)

Новости в JDBC™ 4.1 (Java 7): RowSets (2)

```
try {
        RowSetFactory rsFactory = RowSetProvider.newFactory();
        JdbcRowSet rowSet = rsFactory.createJdbcRowSet();
        rowSet.setUrl("jdbc:mysql://localhost:3306/java21");
        rowSet.setUsername(username);
        rowSet.setPassword(password);
        rowSet.setCommand("SELECT * FROM product");
        rowSet.execute();
        rowSet.absolute(3);          // Позиционира на третия запис
        rowSet.updateFloat("price", 18.70f); //Променя цената на 18.70
        rowSet.updateRow();          // Активира промените в RowSet-а
}
```

Новости в JDBC™ 4.1 (Java 7): RowSets (3)

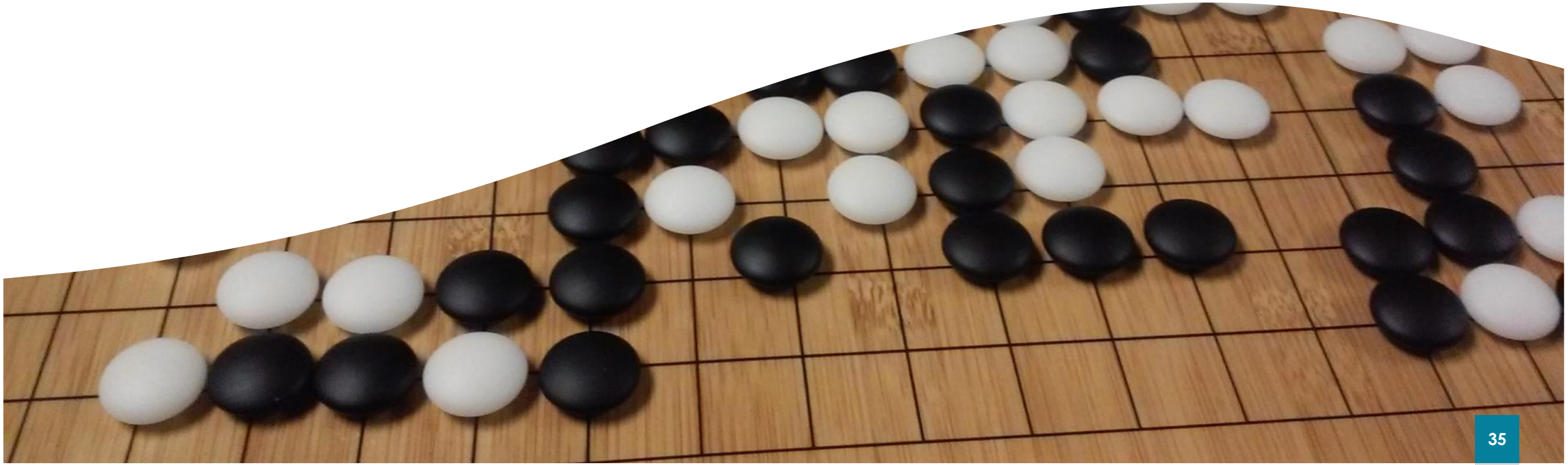
```
try {
        RowSetFactory rsFactory = RowSetProvider.newFactory();
        CachedRowSet rowSet = rsFactory.createCachedRowSet();
        rowSet.setUrl("jdbc:mysql://localhost:3306/java21");
        rowSet.setUsername(username);
        rowSet.setPassword(password);
        rowSet.setCommand("SELECT * FROM product");
        int [] keyColumns = {1}; rowSet.setKeyColumns(keyColumns);
        rowSet.execute();
        rowSet.absolute(3);  rowSet.updateFloat("price", 18.70f);
        rowSet.updateRow();
        rowSet.acceptChanges(con); // Синхронизация с БД
}
```

Литература и интернет ресурси

- Wikipedia – Free Online Enciclopedia – http://wikipedia.org/
- Екел, Б., Да мислим на JAVA. Софтпрес, 2001.
- Oracle® Java™ Technologies webpage – http://www.oracle.com/technetwork/java/
- Oracle®: The Java Tutorials: Lesson: JDBC Basics – http://docs.oracle.com/javase/tutorial/jdbc/basics/index.html
- Oracle®: Новости в JDBC™ 4.1 – http://docs.oracle.com/javase/7/docs/technotes/guides/jdbc/jdbc_41.html
- Joshua Bloch: Automatic Resource Management (V.2) – https://docs.google.com/View?id=ddv8ts74_3fs7483dp

Generics

Practical Exercises



Parameterizied Types: Generics (1)

- Collections and their methods before Java 5 were limited to handle a single type of elements.

- If we want to create typed containers we had to implement different container types for each entity type.

- *Example:* In a e-Bookstore we want to sell **Books** and want the container to contain only **Books** (being strongly typed) --> we should implement separate class **BookList**, as well as for each **Book** we want to keep a list of **Authors** --> we should implement **AuthorList** too, and so on.

Parameterizied Types: Generics (2)

- *Solution:* We can skip writing multiple similar classes (e.g. typed containers for each type of elements) using **Generic types**

- *Generic type invocation:*

```
List<Book> books = new ArrayList<Book>()
List<Author> authors = new ArrayList<Author>()
```

- **<>** – ***Diamond*** *operator* – new in Java™ 7, allows automatic inference of the generic type:

```
List<Book> books = new ArrayList<>()
List<Author> authors = new ArrayList<>()
```

Parameterizied Types: Generics (3)

- *Generic type declaration:*

```
public class Position<T extends Product> {
        private T product;
        ...
        public Position(T product, double quantity) {
                this.product = product;
                this.quantity = quantity;
                price = product.getPrice();
        }
        public T getProduct() {
                return product;
        }
...
```

Generic data type

Conventions Naming Generic Parameters

- **Generic parameters naming conventions:**

T – type parameter (if there are more – S, U, V, W ...)
E – element of a collection – e.g.: List<E>
K – key in associative pair – e.g.: Map<K,V>
V – value in associative pair – e.g.: Map<K,V>
N – number value

- *Example:*

```
public class Invoice <T extends Product> {
        ...
        private List<Position<T>> positions = new ArrayList<>();
    ...
}
```

Generic Methods (1)

- We can implement generic methods and constructors too:

```
public static <U extends Product> String
getPositionsAsString (List<Position<U>> positions) {
      StringBuilder posStr = new StringBuilder();
      int n = 0;
      for(Position<U> p: positions){
          posStr.append( String.format(
              "\n| %1$3s | %2$30s | %3$6s | %4$4s | %5$6s |%6$8s |",
              ++n, p.getProduct().getName(), p.getQuantity(),
              p.getProduct().getMeasure(),p.getPrice(), p.getTotal()
         ));
      }
      return posStr.toString();
} ...
```

Generic Methods (2)

- Invoking generic method / constructor:

```
result += Invoice.<T> getPositionsAsString(positions);
```

- OR we can let Java to automatically infer the generic type:

```
result += Invoice.getPositionsAsString(positions);
```

Bounded Type Parameters

- We can define upper bound constraint for the possible types that can be allowed as actual generic type parameters of the class / method /constructor:

```
public static <U extends Product> String
getPositionsAsString (List<Position<U>> positions) { ... }
```

- OR

```
public static <U extends Product & Printable> String getPositionsAsString
(List<Position<U>> positions) {
    ...
    p.getProduct().print();
    ...
}
```

Generics Sub-typing

- If the class Product extends class Item, can we say that List<Product> extends List<Item> too? Can we substitute the first with the second?
- The answer is „**NOT**“, because the basic generic type is not designed to reflect the specifics of of the Products.
- Dos and donts when using generics inheritance:

```
interface Service extends Item; Service s = new Service( ...);
Collection<Service> services =  ...; services.add(s); // OK
interface Product extends Item; Product p = new Product( ...);
Collection<Product> products =  ...; products.add(p); // OK
Collection<Item> items = ...; items.add(s); items.add(p); // OK
items = products; // NOT OK
items = services; // NOT OK
```

Using ? as Type Specifier (Wildcards)

- If we want to declare that we expect specific, but not pre-determined type, which for example extends the class **Item**, we could use **?** To designate this:

```
Collection<? extends Item> items; // Upper bound is Item
items = products; // OK
items = services; // OK
Items.add(p); // NOT OK – Can not write into it – it is not safe!
Items.add(s); // NOT OK – Can not write into it – it is not safe!
for(Item i: items) { // OK – Can read it – it is known to be at least Item.
    System.out.println( i.getName() + „:“ + i.getPrice() );
}
List<? super Product> products; // Lower bound is Product
products.add(p); // OK – Can write into it – it is now safe.
Product p = products.get(0); //NOT OK may be superclass of Product
```

- Producer extends and Consumer super (PECS) principle

Type Erasure & Reification

- **Type Erasure** – chosen in java as backward-compatibility alternative – information about generic type parameters is erased during compilation, and is NOT available in runtime – the generic type becomes compiled to its basic raw type:

  Collection<Product> products; --(runtime)--> Collection products;

  This design decision creates problems if we want to create generic type instance with **new**, or to convert to the generic type, or to check the generic type using **instanceof**.

- **Reification** – better alternative strategy, implemented in languages such as C++, Ada и Eiffel, using which the generic type information is accessible in runtime.

Generic Containers

- Allow compile time type checking – earlier error detection
- Remove unnecessary typecasting to more specific types – less ClassCastExceptions
- Examples:

```
Collection <String> s = new ArrayList <String>();
Map <Integer, String> table = new HashMap <Integer, String>()
```

- New for loop – for each element of a Collection :

```
for(String i: s) { System.out.println(i) }
```

Main Implementing Classes. Examples

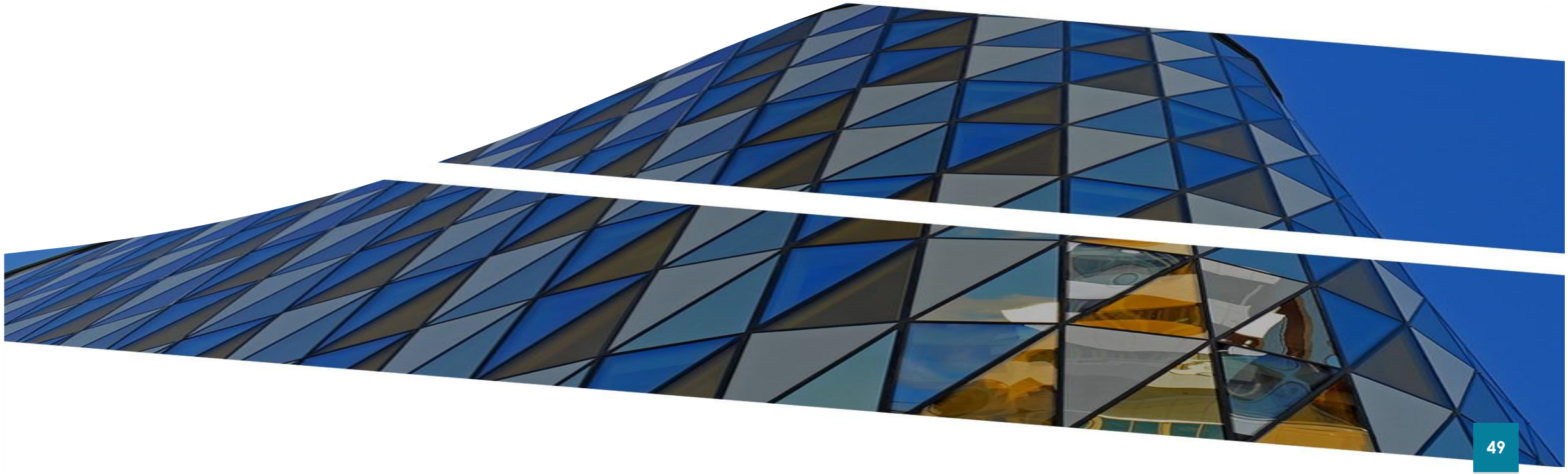
- Associative lists (dictionaries) – interface **Map**
- Comparing different implementations:
  - **HashMap**
  - **TreeMap**
  - **LinkedHashMap**
  - **WeakHashMap**
- Hashing.
- Cash implementations – **Reference, SoftReference, WeakReference и PhantomReference**
- Choosing a container implementation

Литература и интернет ресурси

- Oracle Generics tutorial – https://docs.oracle.com/javase/tutorial/extra/generics/index.html

Java 8 Stream API

Practical Exercises – Functional Programming Koans



Agenda for This Session

- Fundamentals
- Functional interfaces
- Method references
- Constructor references

Новости в Java™ 8

- Ламбда изрази и поточно програмиране – пакети **java.util.function** и **java.util.stream**)
- Референции към методи
- Методи по подразбиране и статични методи в интерфейси – множествено наследяване на поведение в Java 8
- Функционално програмиране в Java 8 с използване на монади (напр. Optional, Stream) – предимства, начин на реализация, основни езикови идиоми, примери

Функционални интерфейси в Java™ 8

- Функционален интерфейс = интерфейс с един абстрактен метод
  SAM (Single Abstract Method) – @FunctionalInterface
- Примери за функционални интерфейси в Java 8:

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}
public interface ActionListener extends EventListener {
    public void actionPerformed(ActionEvent e);
}
public interface Runnable {
    public void run();
}
public interface Callable<V> {
    V call() throws Exception;
}
```

Ламбда изрази – пакет java.util.function

Примери:

```
(int x, int y) -> x + y

() -> 42

(a, b) -> a * a + b * b;

(String s) -> { System.out.println(s); }

book -> book.getAuthor().fullName()

voter -> voter.getAge() >= legalAgeOfVoting

(person1, person2) -> person1.getAge() - person2.getAge()

(song1, song2) -> song1.getArtist().compareTo(song2.getArtist())
```

Правила за форматирне на ламбда изрази

- **Ламбда изразите (функциите)** могат да имат произволен брой **параметри**, които се ограждат в скоби, разделят се със запетаи и могат да имат или не деклариран тип (ако нямат - типът им се извежда от **контекста на използване = target typing**). Ако са само с един параметър, то скобите не са задължителни.
- **Тялото на ламбда изразите** се състои от произволен езикови конструкции (statements), разделени с **;** и заградени във фигурни скоби. Ако имаме само една езикова конструкция – израз то използването на фигурни скоби не е необходимо – в този случай стойността на израза автоматично се връща като стойност на функцията.

Пакет java.util.function

- **Predicate<T>** – предикат = булев израз представящ свойство на обекта подавано като аргумент
- **Function<A,R>**: функция която приема като аргумент **A** и го трансформира в резултат **R**
- **Supplier<T>** – с помощта на **get()** метод всеки път връща инстанция (обект) – фабрика за обекти
- **Consumer<T>** – приема аргумент (метод **accept()**) и изпълнява действие върху него
- **UnaryOperator<T>** – оператор с един аргумент **T -> T**
- **BinaryOperator<T>** – бинарен оператор **(T, T) -> T**

Поточно програмиране (1)

Примери:

```
books.stream().map(book ->
    book.getTitle()).collect(Collectors.toList());
books.stream()
       .filter(w -> w.getDomain() == PROGRAMMING)
       .mapToDouble(w -> w.getPrice()) .sum();
document.getPages().stream()
     .map(doc -> Documents.characterCount(doc))
     .collect(Collectors.toList());
document.getPages().stream()
     .map(p -> pagePrinter.printPage(p))
     .forEach(s -> output.append(s));
```

Поточно програмиране (2)

Примери:

```
document.getPages().stream()
        .map(page -> page.getContent())
        .map(content -> translator.translate(content))
        .map(translated -> new Page(translated))
        .collect(Collectors.collectingAndThen(
                Collectors.toList(),
                pages -> new
        Document(translator.translate(document.getTitle()), pages)));
```

Референции към методи

- Статични методи на клас – Class::staticMethod
- Методи на конкретни обектни инстанции – object::instanceMethod
- Методи на инстанции реферирани чрез класа – Class::instanceMethod
- Конструктори на обекти от даден клас – Class::new

```
Comparator<Person> namecomp = Comparator.comparing(Person::getName);

Arrays.stream(pageNumbers).map(doc::getPageContent).forEach(Printers::print);

pages.stream().map(Page::getContent).forEach(Printers::print);
```

Статични и Default методи в интерфейси

- Методите с реализация по подразбиране в интерфейс са известни още като virtual extension methods или defender methods, защото дават възможност интерфейсите да бъдат разширявани, без това да води до невъзможност за компилация на вече съществуващи реализации на тези интерфейси (което би се получило ако старите реализации не имплементират новите абстрактни методи).
- Статичните методи дават възможност за добавяне на помощни (utility) методи – например factory методи директно в интерфейсите които ги ползват, вместо в отделни помощни класове (напр. Arrays, Collections).

Пример за default и static методи в интерфейс

- @FunctionalInterface

```
interface Event {
    Date getDate();
    default String getDateFormatted() {
            return String.format("%1$td.%1$tm.%1$tY", getDate());
    }
    public static <T, U extends Comparable<? super U>>
    Comparator<T> comparing(Function<T, U> getKey) {
        return (c1, c2) -> getKey.apply(c1).compareTo(getKey.apply(c2));
    }
}
Event current = () -> new Date();
System.out.println(current.getDateFormatted());
```

Функционално програмиране и монади

- Понятие за **монада** във функционалното програмиране (теория на категориите) – **Монадата** е множество от три елемента:

- Параметризиран тип **M<T>**
- "**unit**" функция: **T -> M<T>**
- "**bind**" операция: **bind(M<T>, f:T -> M<U>) -> M<U>**

- В Java 8 пример за монада е класът **java.util.Optional<T>**

Параметризиран тип: **Optional<T>**

- "**unit**" функции: **Optional<T> of(T value) , Optional<T> ofNullable(T value)**
- "**bind**" операция: **Optional<U> flatMap(Function<? super T,Optional<U>> mapper)**

Exercise: Java 8 Functional Programming Koans

Available @GitHub: https://github.com/iproduct/course-java-web-development/tree/master/lambda-tutorial-master

1. Read carefully the JavaDoc for the unit tests stating the problem to solve: Exercise_1_Test.java, Exercise_2_Test.java, Exercise_3_Test.java, Exercise_4_Test.java and Exercise_5_Test.java
2. Fill the code in place of comments like: // [your code here]
3. Run the unit tests to check if your proposed solution is correct. If not return to step 1.

Литература и интернет ресурси

- Oracle tutorial – lambda expressions - http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html
- Java SE 8: Lambda Quick Start - http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart/index.html
- OpenJDK Lambda Tutorial - https://github.com/AdoptOpenJDK/lambda-tutorial

Thank's for Your Attention!

Trayan Iliev

IPT – Intellectual Products & Technologies

http://iproduct.org/

http://robolearn.org/

https://github.com/iproduct

https://twitter.com/trayaniliev

https://www.facebook.com/IPT.EACAD